Automating Inference of OCL Business Rules from User Scenarios

Duc-Hanh Dang, Jordi Cabot

Automating Inference of OCL Business Rules from User Scenarios

Duc-Hanh Dang*† and Jordi Cabot*
*AtlanMod, École des Mines de Nantes - INRIA, LINA, Nantes, France
{duc-hanh.dang|jordi.cabot}@inria.fr
†VNU-University of Engineering and Technology, Hanoi, Vietnam
hanhdd@vnu.edu.vn

***Abstract*—User Scenarios have been advocated as an effective means to capture requirements by describing the system-to-be at the instance or example level. This instance-level information is then used to infer a possible software specification consistent with the provided valid and invalid scenarios. So far existing approaches have often focused on the generation of static models but have omitted the inference of business rules that could complement the static models and improve the precision of the software specification. In this sense this paper provides a first set of invariant inference patterns that are applied on valid and invalid snapshots in order to generate OCL (Object Constraint Language) integrity constraints that the system should always satisfy. We strengthen the confidence of inferred results based on the user's feedback of generated examples and counterexamples for the considered constraint. The approach is realized with a prolog-based tool that could support the designer to effectively define OCL integrity constraints in a semi-automatic way.**

***Keywords*-User Scenarios, OCL, Invariants, Business Rules, Snapshots, Prolog, Patterns.**

I. Introduction

User Scenarios have been advocated as an effective means to capture requirements by describing the system-to-be at the instance or example level [1]. This instance-level information is then used to infer a possible software specification consistent with the provided valid and invalid scenarios. So far existing approaches have often focused on the generation of static models but have omitted the inference of business rules that could complement the static models and improve the precision of the software specification. This puts forward a need to infer business rules from user scenarios.

Business rules are a means to reflect business aspects of a system [2], [3]. Currently, business rules are often defined by an implicit translation from a description in natural language to another specification either in the Object Constraint Language (OCL) [4], [5] or in semi-natural languages like SBVR (Semantics of Business Vocabulary and Business Rules) [3] and RuleSpeak [6]. Business rules in OCL are often used as OCL integrity constraints that the system should always satisfy.

The essence of the inference of OCL business rules from scenarios is to consider the relationship between snapshots (system states, represented by instances of a conceptual model) and OCL invariants. Several authors offer methods to check if a snapshot is valid [7]. The paper in [8] offers a technique to recheck snapshots incrementally as they change. Current works often translate OCL specifications into the other specification environments such as CSP [9], Alloy [10], relational calculus [11], description logic [12] and relational logic [13] in order to effectively validate snapshots or to find a valid (invalid) snapshot or to model-check properties. The paper in [14] proposes to generate OCL constraint templates by taking a lexical analysis. To the best of our knowledge, there is only the work in [15] focusing on such an inference of OCL invariants. That work proposes using genetic programming in order to automatically discover well-formedness rules in OCL for metamodels from valid and invalid model examples.

In this paper, we propose to employ a pattern-based inference mechanism in order to infer OCL business rules from user scenarios. First, OCL invariant patterns are defined as inference patterns that could be applied on a set of valid and invalid snapshots in order to generate a corresponding OCL invariant. Second, a prolog-based algorithm with the support of the solver ECL^{i}PS^e [16] allows us to fetch appropriate patterns from an available pattern catalog and to obtain result sets of OCL invariants corresponding the provided valid and invalid snapshots. Then, we define the confidence of the inferred result based on the user's feedback of generated examples and counterexamples for the considered constraint. In that way we could offer a tool support for the designer to effectively define OCL business rules in a semi-automatic way.

The remainder of this paper is organized as follows. Section II presents a motivating example of inferring OCL business rules. Section III overviews our approach. Section IV first explains the basic idea of OCL invariant patterns, then presents a prolog-based technique to manipulate patterns. Section V introduces a method with a prototype tool to infer OCL business rules. The section continues with a discussion. Section VI surveys related work. This paper is closed with conclusions and an outlook on future work.

II. Motivating Example

This section focuses on the context where system specifications including use case models, conceptual models and other structural and behavioral models are basically generalized from User Scenarios. We point out that specifications with business rules, that appear as OCL restrictions on conceptual models, should also be inferred from scenarios.

A. From User Scenarios to OCL Business Rules

Scenarios describe the information of a system at the instance or example level. Both valid and invalid scenarios can be provided, e.g., the valid case "*Peter works for the Department 3*" and the invalid one "*Peter is a manager of himself*". Such an instance level information is then generalized into models and specifications that are used in software development. For example, from such example scenarios that might help us to infer the conceptual model as shown in Fig. 1.

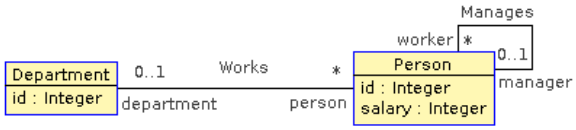


Fig. 1. A conceptual model in form of class diagram.

Conceptual models often need to be complemented with restrictions in order to improve the precision of the software specification. We might find the following restrictions on the example conceptual model: "*the salary of a person is always less than 100*" and "*a manager never manages her- or himself*". Such information of business aspects of the system is often described by business rules [2], [3].

The declarative language OCL, that is based on first-order predicate logic, was invented in order to express business rules [4], [5], [17]. OCL business rules as *OCL invariants* are conditions that the system should always satisfy. They specify restrictions on conceptual models, e.g., the example restriction "*the salary of a person is less than 100*" could be expressed in OCL: `context Person inv: self.salary < 100`. Note that an OCL condition is often defined in the context of an instance of a class, that we might access with a so-called *self* variable. OCL could be employed in several other cases such as (1) to describe pre- and post conditions of operations, (2) to give restrictions as guards in a state transition system and (3) to query over a given system state with OCL queries [4].

B. Challenge to Infer OCL Business Rules

In principle, we could obtain OCL business rules from a specification in a semi-natural language like SBVR [3] and RuleSpeak [6]. However, such a transformation is obviously hard [18] and as far as we know, there is currently no effective solution for it [19]. Within the context modeling with User Scenarios, OCL business rules should also be inferred from the instance-level information of the system. However, this is also a challenge. Consider the domain as represented in Fig. 1, it might require business rules as follows.

1) A person has a unique identifier:
```
context Person inv uniqueId:
Person::allInstances() -> forAll(p1, p2 |
p1<>p2 implies p1.id<>p2.id)
```
2) A person's salary is greater than 20 and less than 100:
```
context Person inv salaryInterval:
20 < self.salary and self.salary < 100
```
3) A department always has more than two persons:
```
context Department inv personCard:
self.person->size() > 2
```
4) A manager never manages him- or herself:
```
context Person inv nonSelfManager:
self.manager->excludes(self)
```
5) If the salary of a person of a department is less than 70, the department id is less than 5:
```
context Person inv salary_id:
self.salary < 70 implies self.department.id < 5
```

The challenging question is that how such business rules could be inferred from user scenarios.

III. Basic Idea

This section overviews our approach to inferring OCL business rules from user scenarios.

A. From Scenarios to Snapshots

Snapshots are possible instances of a conceptual model. They are often visualized by *object diagrams*, that conform to a *class diagram* and are often restricted by *OCL conditions* [20]. Specifically, a snapshot includes a set of *objects* that are connected to each other by *links*. Figure 2 illustrates two snapshots that conform to the class diagram shown in Fig. 1.

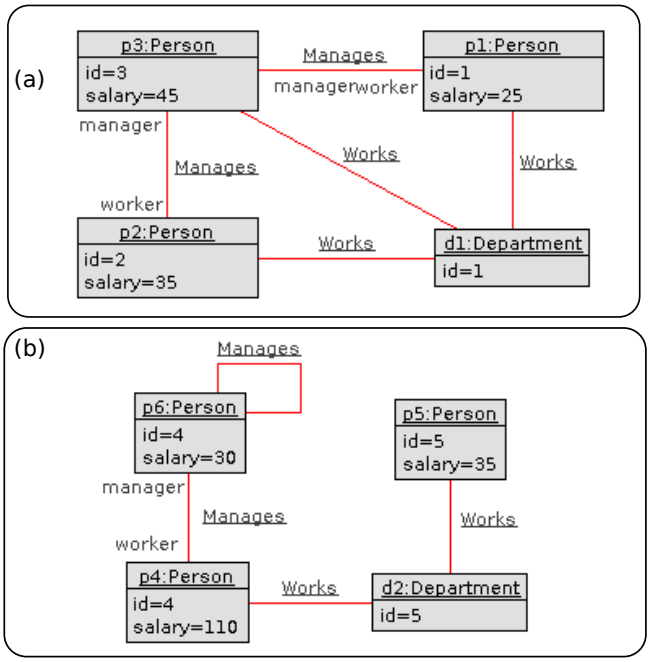


Fig. 2. Example valid and invalid snapshot.

Scenarios are often represented as a sequence of snapshots. In this sense a snapshot might be valid or invalid. For example, with the business rule "*the salary of a person is always greater than 20 and less than 100*", the snapshot (a) is valid, and the snapshot (b) is invalid. Note that the snapshot (b) is even rejected by each example invariant as mentioned in SubSect. II-B.

B. Overview of the Approach

Figure 3 illustrates for our approach: The *Conceptual Model* represents the underlying domain model as a class diagram. The *Domain Knowledge*, that could be domain experts or model-generating tools such as USE [7] and UML2CSP [9], provides the input *Valid and Invalid Snapshots* represented by object diagrams that conform to the conceptual model.

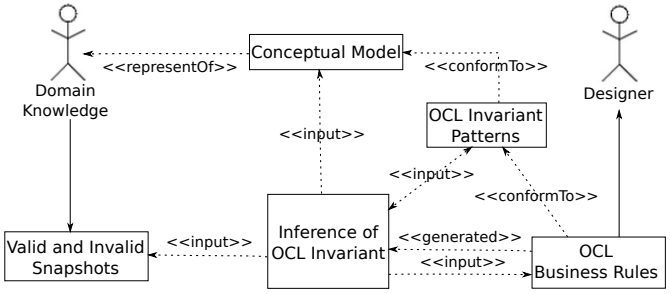


Fig. 3. Overview of the Approach.

The *OCL Invariant Inference* analyses the input snapshots and selects relevant patterns from the *OCL Invariant Patterns* in order to form OCL invariants. The *OCL Business Rules* returns the inferred invariants to the *Designer*.

IV. Pattern-Based Inference Mechanism

We propose a pattern-based inference mechanism in order to infer OCL invariants. Within this mechanism OCL invariant patterns are encoded as prolog predicates. To prove them means to match patterns in order to generate OCL invariants. We illustrate the mechanism with a first set of patterns corresponding to OCL invariants that often occur in practice.

A. OCL Invariant Pattern

We employ OCL invariant patterns as templates in order to generate OCL invariants. For example, the invariant

```
context Person inv: self.manager->excludes(self)
```

could be generated by such a template

```
context [A] inv: self.[role]->excludes(self).
```

The template for the invariant

```
context Person inv:
20 < self.salary and self.salary < 100
```

should be the one

```
context [A] inv:
[left] < self.[attr] and self.[attr] < [right].
```

To apply a pattern on pair of valid and invalid snapshot sets means to assign values to variables that occur in the OCL template in order to obtain an invariant that accepts the valid snapshot set and rejects the invalid one. The process in general includes two steps. First, we assign the *[A]* variable to a class name corresponding to the context of the OCL expression, and then assigning values to the other variables that contain class, attribute, operation and role names. We refer them as *class variables*. Second, we analyze the input snapshot sets in order to define values for the remaining variables, e.g., the *[left]* and *[right]* in the template for the second example invariant. The variables are referred to as *data variables*.

Definition 1. (*OCL Invariant Pattern*) Let $CD \in \mathcal{CD}$ be a class diagram, $CLASS$ the set of its classes, and $SNAPSHOT$ the set of its snapshots. Let be given [1] $\tau : \mathcal{P}(SNAPSHOT) \times \mathcal{P}(SNAPSHOT) \rightarrow \mathcal{P}(String^* \times D^*)$, $\varphi : String^* \times D^* \rightarrow String$ such that $\tau(sOK, sNOK) = \{para = ((A, c_1, ..., c_n), (v_1, ..., v_m))\}$, and $\varphi(para)$ returns a parameterized OCL invariant expression of the class $A \in CLASS$, where $\{c_i\}$ are class variables, and $\{v_i\}$ are data variables of the OCL expression. The tuple (φ, τ) is referred to as *an OCL invariant pattern*, the φ as *the OCL template function*, and the τ as *the matching function* of the pattern. The input snapshot set sOK ($sNOK$) is referred to as *the set of valid (invalid) snapshots*.

Figure 4 depicts the OCL invariant pattern of the *NonSelfInclusion* pattern. This pattern represents for OCL invariants that forbid an object to link to itself in an association. Note that the classes A and B may coincide to each other. This pattern is presented by the tuple $(\varphi((A, role), [\,]), \tau)$, where the $A, role \in String$ refer to the name of a class and its role. The $\varphi((A, role), [\,])$ returns an invariant as shown in Fig. 4.

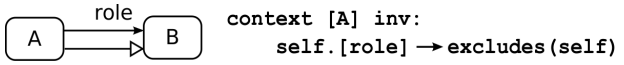


Fig. 4. The *NonSelfInclusion* pattern is $(\varphi((A, role), [\,]), \tau)$, where $A, role \in String$ refer to the name of a class and its role, that are based on input snapshots and defined by the matching function τ.

We propose a prolog-based method to specify matching functions $\tau(sOK, sNOK)$. The basic idea is to map the parameterized OCL invariant of a pattern to a predicate in prolog: The predicate (goal) is proved if and only if the corresponding snapshot is valid, i.e., accepted by the OCL invariant. For a detailed explanation how to encode OCL invariants in prolog, we refer the reader to the work in [9]. We could define the matching function of the *NonSelfInclusion* pattern as follows:

```
%--Applying the NonSelfInclusion Pattern-----------
apply_nonSelfInc(SOK, SNOK, Para):-
  Para = [X_Class,X_Role],
  %--Constraints on class variables----------------
  LocalPara = [X_Class,X_Assoc,X_Role,Role2],
  roleType(X_Assoc,Role2,X_Class),
  roleType(X_Assoc,X_Role,ClsB),
  X_Role \= Role2,
  %--Ensuring X_Class is a subtype of ClsB---------
  ( X_Class \= ClsB
    ->( aux_subTypeList(ClsB, TypeList),
        member(X_Class, TypeList)
      );true
  ),
  %--Ensuring the invariant accepts SOK------------
  ( foreach(SnapshotOK, SOK),
    param(LocalPara)
    do
      nonSelfInc(SnapshotOK, LocalPara)
  ),
```

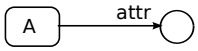
[1] $\mathcal{P}(S)$ is the power set of S

```
  %--Ensuring the invariant rejects SNOK-----------
  ( foreach(SnapshotNOK, SNOK),
    param(LocalPara)
    do
      (not nonSelfInc(SnapshotNOK, LocalPara))
  ).
%--Encoding the OCL template in prolog-------------
%--in order to check if a snapshot is valid--------
nallInstances1nonSelfInc(Instances,[Para],Result):-
  Para = [X_Class,_,_,_],
  ocl_allInstances(Instances, X_Class, Result).
nVariable2nonSelfInc(_, Vars, Result):-
  ocl_variable(Vars,1,Result).
nNavigation3nonSelfInc(Instances, Vars, Result):-
  append(_, [Para], Vars),
  Para = [_,X_Assoc,X_Role,Role2],
  nVariable2nonSelfInc(Instances, Vars, Value1),
  ocl_navigation(Instances,X_Assoc,Role2,
                          X_Role,Value1, Result).
nVariable4nonSelfInc(_, Vars, Result):-
  ocl_variable(Vars,1,Result).
nexcludes5nonSelfInc(Instances, Vars, Result):-
  nNavigation3nonSelfInc(Instances,Vars,Value1),
  nVariable4nonSelfInc(Instances, Vars, Value2),
  ocl_set_excludes(Value1, Value2, Result).
nforAll6nonSelfInc(Instances, Vars, Result):-
  nallInstances1nonSelfInc(Instances,Vars,Value1),
  ocl_col_forAll(Instances, Vars, Value1,
                  nexcludes5nonSelfInc, Result).
nonSelfInc(Instances, Para):-
  nforAll6nonSelfInc(Instances, [Para], Result),
  Result #=1.
```

In this definition the OCL invariant pattern corresponds to a predicate $nonSelfInc(Instances, Para)$, that checks if the snapshot $Instances$ is valid. By proving the $apply\_nonSelfInc(SOK, SNOK, Para)$ predicate, we could obtain a $Para$ value such that the invariant accepts all snapshots SOK and rejects $SNOK$. The $Para$ value allows us to define a corresponding OCL invariant using the template `context [X_Class] inv: self.[X_Role]->excludes(self).`

B. Presentation of OCL Invariant Patterns

We propose to describe OCL invariant patterns in a common template, illustrated with the *Interval* pattern as follows:




```
context [A] inv:
  (left1 < self.[attr] and self.[attr] < right1)
  or ...
  (leftn < self.[attr] and self.[attr] < rightn)
```

Fig. 5. The *Interval* pattern is the tuple $(\varphi((A, attr), (l_1, r_1, ..., l_n, r_n)), \tau)$, where the A and $attr$ refer to the name of a class and its attribute, the matching function τ defines intervals as the input for the OCL template φ.

1) *Name.* The pattern is referred to as an *Interval* pattern.
2) *Description.* This pattern aims to generate OCL invariants as a restriction on an integer attribute of a class.
3) *OCL Template.* This pattern is the tuple $(\varphi((A, attr), (l_1, r_1, ..., l_n, r_n)), \tau)$, where the $A, attr \in String$ refer to the name of a class and its attribute, and $(l_1, r_1, ..., l_n, r_n) \in Integer^+$ to define intervals for the $attr$ attribute. The OCL template is illustrated as in Fig. 5.
4) *Pattern Matching.* The matching function of this pattern could be encoded in prolog such that $\tau(sOK, sNOK) = \{((A, att), f(|sOK|, |sNOK|))\}$, where the A, $attr$ are defined by the input class diagram CD. The list $|sOK|, |sNOK| \in Integer^+$ are obtained by flattening $sOK, sNOK \subset \mathcal{P}(Integer^+)$.

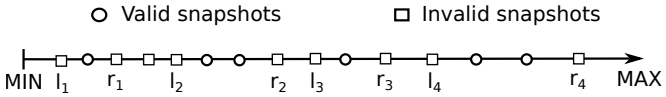


Fig. 6. Illustration for the interval function.

The function $f : \mathcal{P}(Integer) \times \mathcal{P}(Integer) \rightarrow Integer^+$ returns intervals $(l_1, r_1, ..., l_n, r_n)$ as illustrated in Fig. 6. Such a f function could be encoded in prolog with the following specification: $f(|sOK|, |sNOK|) = (l_1, r_1, ..., l_n, r_n)$ iff $\forall 1 \leq i \leq n$

- $l_i, r_i \in |sNOK| \cup \{MIN, MAX\}$,
- $MIN \leq l_i < r_i < l_{i+1}$, where $l_{n+1} = MAX$,
- $\nexists s \in |sOK|, r_i \leq s \leq l_{i+1}$,
- $\exists s \in |sOK|, l_i < s < r_i$.

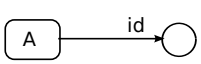
5) *Example.* The input includes (1) the class $A = 'Person'$ and the integer attribute $attr = 'salary'$ as depicted in Fig. 1, and (2) the valid and invalid snapshot set $|sOK| = \{45, 89\}, |sNOK| = \{20, 100\}$. The function $f_i(|sOK|, |sNOK|)$ returns $(l_1 = 20, r_1 = 100)$. The OCL template allows us to generate the invariant: `context Person inv: 20 < self.age and self.age < 100.`

C. First Set of OCL Invariant Patterns

Table I shows a first set of OCL invariant patterns for typical OCL invariants that often occur in practice [21], [4], [5]. We might find examples for them with the invariants mentioned in SubSect.II-B. We briefly introduce the patterns as follows.

TABLE I
A FIRST SET OF OCL INVARIANT PATTERNS.

Id	Name	Description
01	UniqueAttribute	To restrict on the uniqueness of an attribute
02	Interval	To restrict on an attribute
03	Multiplicity	To restrict on the multiplicity in an association
04	NonSelfInclusion	To forbid an object links to itself
05	AttributeRelation	To restrict on two attributes of two classes

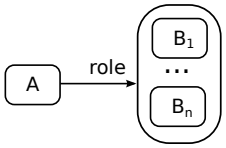



```
context [A] inv:
 [A]::allInstances()->forAll(p1,p2 |
  p1<>p2 implies p1.[id] <>p2.[id])
```

Fig. 7. The *UniqueAttribute* pattern is $(\varphi((A, id), [\,]), \tau)$, where $A, id \in String$ refer to the name of a class and its unique attribute.

Figure 7 shows the OCL template of the *UniqueAttribute* pattern. This pattern allows us to generate an OCL restriction on the uniqueness of an attribute of a class. Its matching

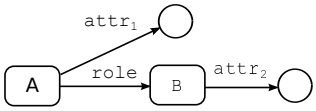
function could be encoded in prolog in a similar way to the *NonSelfInclusion* pattern as mentioned in SubSect. IV-A.




```
context [A] inv:
  (l1 < self.[role]->size()and self.[role]->size() < r1)
  or ...
  (ln < self.[role]->size()and self.[role]->size() < rn)
```

Fig. 8. The *Multiplicity* pattern is $(\varphi((A, role), [\,]), \tau)$, where $A, role \in String$ refer to the name of a class and its role.

Figure 8 illustrates for the OCL template of the *Multiplicity* pattern. This pattern generates OCL invariants as restrictions on the multiplicity of participants in an association. Its matching function is similar to the one of the *Interval* pattern.




```
context [A] inv:
  self.[attr1]< p implies self.[role].[attr 2] < q
```

Fig. 9. The $(\varphi((A, role, attr_1, attr_2), (p, q)), \tau)$ is referred to as the *AttributeRelation* pattern, where $A, role, attr_1, attr_2 \in String$ refer to the name of a class, its role name, its attribute, and the attribute of the associated class. The τ defines the input for φ using a f_i function $\mathcal{P}(Integer \times Integer) \times \mathcal{P}(Integer \times Integer) \rightarrow Integer \times Integer$.

Figure 9 presents the OCL template of the *AttributeRelation* pattern. This pattern generates OCL invariants to restrict the relation between two integer attributes of two class associated to each other. The matching function τ could be encoded in prolog such that $\tau(sOK, sNOK) = \{((A, role, attr_1, attr_2), f(|sOK|, |sNOK|))\}$, where the $A, role, attr_1, attr_2$ are defined by the input class diagram CD. The $|sOK|, |sNOK| \in \mathcal{P}(Integer \times Integer)$ are obtained from $sOK, sNOK \subset \mathcal{P}(Integer \times Integer)$ by a flattening. Such a $f : \mathcal{P}(Integer) \times \mathcal{P}(Integer) \rightarrow Integer^+$ could be encoded in prolog with the following specification: $f(|sOK|, |sNOK|) = (p, q)$ iff

- $p = max\{sNOK_{0i}\} + 1$,
- $q = min\{sNOK_{1i}\}$,
- $\forall i, (sOK_{0i} < p) \wedge (sOK_{1i} < q)$.

D. Prolog-Based Pattern Manipulation

We introduce an algorithm with a simple strategy to select appropriate patterns from an available catalog of patterns $PATTERN$. Basically, as illustrated in Fig. 10, with each selected pattern pat_i we need to define a subset $SNOKpop$ of the input invalid snapshot set $SNOK$ such that it is rejected by the generated invariant. Note that each inferred invariant must accept all valid snapshots in SOK. In that way we could encode the algorithm in prolog as follows.

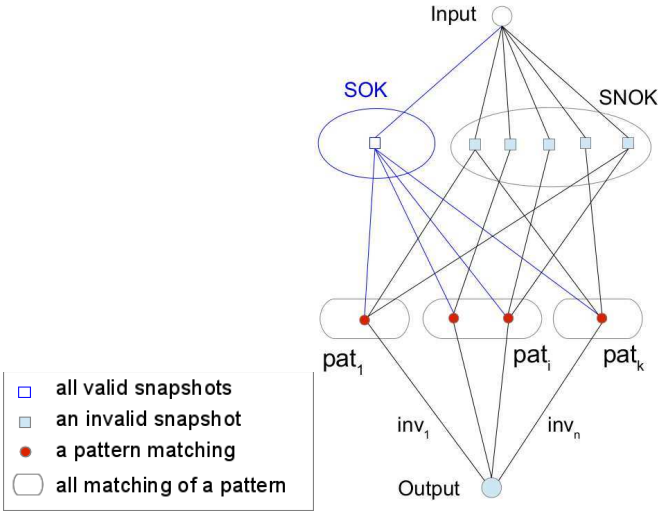


Fig. 10. Pattern-based inference of OCL invariants.

```
apply_part(SOK, SNOK, PATTERN, Pattern, INV) :-
  subset(SNOKpop, SNOK),
  SNOKpop \= [],
  subtract(SNOK, SNOKpop, SNOKrest),
  applyPattern(Pattern, SOK, SNOKpop, Para),
  writeInv(Pattern, Para, Inv),
  apply_all(SOK, SNOKrest, PATTERN, INVrest),
  INV = [Inv | INVrest].

apply_all(SOK, [], PATTERN, [Inv]):-
  member(Pat, PATTERN),
  applyPattern(Pat, SOK, [], Para),
  writeInv(Pat, Para, Inv).

apply_all(SOK, SNOK, PATTERN, INV) :-
  member(Pat, PATTERN),
  member(N,[1,2]),
  ( %--to reject SNOK by the generated invariant-----
    N = 1
       %--if the invariant could reject SNOK---------
    -> ( applyPattern(Pat, SOK, SNOK, Para)
         -> writeInv(Pat, Para, Inv),
            INV = [Inv];
       %--else, it only could reject part of SNOK----
         apply_part(SOK, SNOK, PATTERN, Pat, INV)
       );
    %--the invariant rejects only part of SNOK-------
    apply_part(SOK, SNOK, PATTERN, Pat, INV)
  ).
```

Note that in this specification the predicate $applyPattern(Pat, SOK, SNOK, Para)$ corresponds to the matching function τ of the Pat pattern, and the predicate $writeInv(Pat, Para, Inv)$ corresponds to the template function φ of this pattern.

V. Tool Support and Discussion

This section explains how our approach is realized. We aim to take advantage of the user's feedback of the inferred result in order to eliminate irrelevant invariants and to strengthen the confidence of the inference. In that way, we could obtain the final result as a propagation from such a user interaction. A prototype tool is introduced for the aim.

A. Overview of the Prototype Tool

Figure 11 illustrates for our prototype tool. First, the user loads the input model with a xmi specification file. The model is encoded in prolog based on the UMLtoCSP tool [9]. Then, the *Inference of OCL Invariants* module analyzes the input set of valid and invalid snapshots provided by the user and sends a query to the ECL^{i}PS^e solver in order to infer OCL invariants.

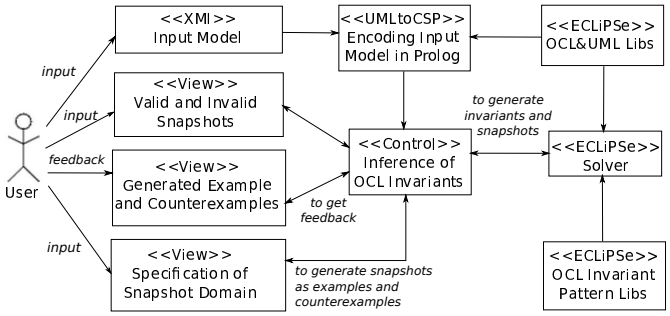


Fig. 11. Overview of the prototype tool to infer OCL invariants.

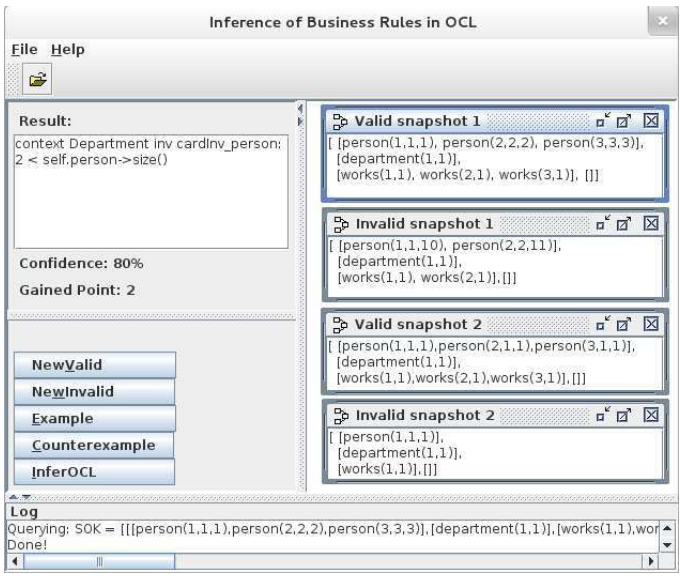


Fig. 12. Example for the inference of OCL invariants.

Based on the domain specification for snapshots provided by the user, this module could require the solver to find snapshots as examples and counterexamples for current invariants. The user's feedback of the relevance of snapshots allows us to update the inferred result by a new and more relevant one.

B. A Method to Strengthen the Confidence of the Inference

In order to illustrate for the method, we focus on the inference of the following invariant:

```
context Department inv personCard:
self.person->size() > 2.
```

We could consider input snapshots as instances of structures for classes and associations and present them in a textual form as depicted in Fig. 12. The underlying structures include `person(oid,id,salary)`, `department(oid,id)`, `works(person,department)`, and `manages(manager,worker)`, that corresponds to the example model shown in Fig. 1. For example, the snapshot (a) in Fig. 2 could be viewed as follows:

```
Sok=[[person(1,1,25),person(2,2,35),person(3,3,45)],
     [department(1,1)],
     [works(1,1),works(2,1),works(3,1)],
     [manages(3,1),manages(3,2)]]
```

The inference scenario starts with the input *Valid snapshot 1* and *Invalid snapshot 1* as shown in Fig. 12. By applying the *Interval* and *Multiplicity* patterns as mentioned in Sect. IV, the inference system could return the two invariants:

- `context Person inv intv_salary:`
  `self.salary < 10`
- `context Department inv personCard:`
  `self.person->size() > 2`

We might define the confidence of the inference based on the total number n of inferred results. In the best case, there is only one inferred result, we assign the confidence to 80%. In the other case, the confidence will be $(100/n)\%$. So, in this case the confidence of the inference is 50%.

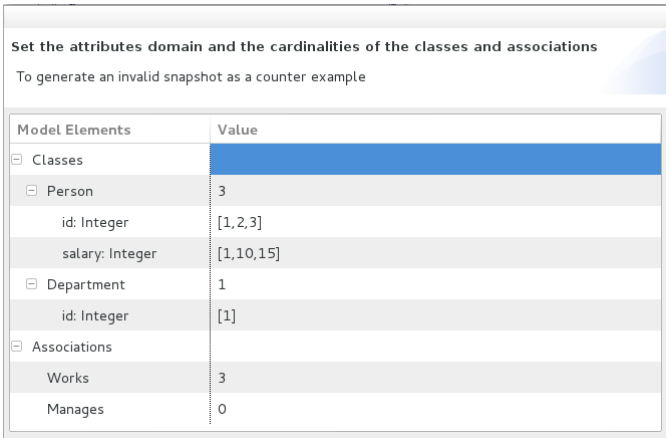


Fig. 13. The user specifies a domain for valid snapshots such that as a counterexample could be found, the considered invariant is irrelevant.

Second, the user needs to provide feedback in order to eliminate the first irrelevant invariant (the one generated by the *Interval* pattern). The user then requires the tool to search a counterexample from a specified domain of snapshots, where all snapshots of the domain are valid as her/his expectation, i.e., the inferred result is irrelevant as a counterexample could be found. Such a domain specification is depicted as in Fig. 13. Consider the first invariant, the tool could return the following counterexample:

```
Snok=[[person(1,1,10),person(2,1,1),person(3,1,1)],
      [department(1,1)],
      [works(1,1),works(2,1),works(3,1)],[]]
```

This indicates the first invariant is irrelevant and eliminated. The confidence of the current inference is updated with 80%.

The user could employ a similar strategy in order to eliminate irrelevant invariants: S/he specifies a domain where all snapshots are invalid as her/his expectation, i.e., the considered result is irrelevant as an example could be found.

At this point the confidence is 80%, i.e., there remains only one inferred result. The user could verify the relevance of the result using generated examples and counterexamples. The gained point as depicted in Fig. 12 records the total number of relevant cases. As an irrelevance occurs, i.e, the current result is irrelevant, the inference is restarted with updated snapshots.

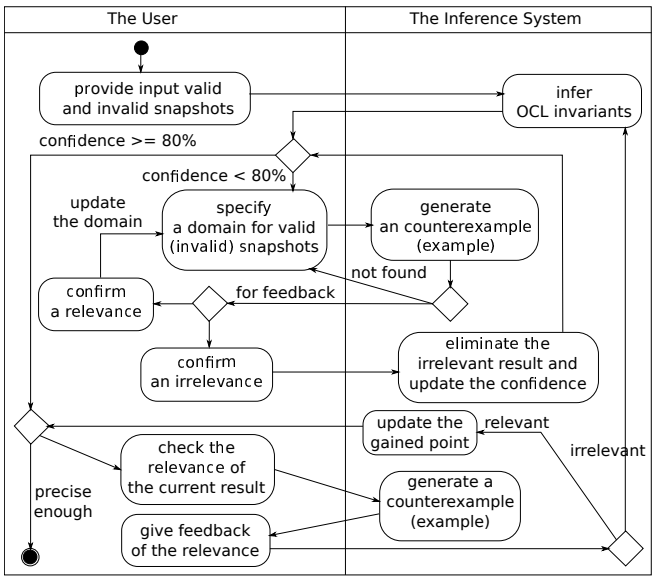


Fig. 14. Method to strengthen the confidence of the inference.

We could summarize the method to strengthen the confidence of the inference with the diagram as shown in Fig. 14.

C. Experiment on the Running Example

This section presents our experiment on the running example. We defined successfully scenarios so that the prototype tool could infer the example OCL invariants. The experimental result is summarized as in Table II. At first, the confidence is often less than 80% as illustrated by the last column of the table. By a propagation from the user's feedback, the tool could return the one final result with the confidence 80% together with the gained point 10 for both examples and counterexamples. At this point we could increase the gained point more for a precise enough result.

D. Discussion

Based on the backtracking feature of prolog, the algorithm mentioned in SubSect. IV-D could provide all results inferred from a set of input snapshots with the support of a catalog of patterns. However, the time complexity of the algorithm is exponential $\theta(2^n)$, where n is the size of input data. This prevents from applying the algorithm in practice. Consider our realization it may get over the concern since the size of input data in this case is often small.

TABLE II
INFERRING OCL INVARIANTS FOR THE RUNNING EXAMPLE.

Name	**Input snapshots**	**1^{st}Step**
uniqueId	Sok=[[person(1,1,10),person(2,2,10)],[],[],[]], Snok=[[person(1,1,10),person(2,1,10)],[],[],[]]	80%
salaryInterval	Sok=[[person(1,1,21),person(2,2,99)],[],[],[]], Snok1=[[person(1,1,20)],[],[],[]], Snok2=[[person(1,1,100)],[],[],[]]	50%
personCard	Sok=[[person(1,1,1),person(2,2,2), person(3,3,3)],[department(1,1)], [works(1,1),works(2,1),works(3,1)],[]] Snok=[[person(1,1,10),person(2,2,11)], [department(1,1)],[works(1,1),works(2,1)],[]]	50%
nonSelfManager	Sok=[[person(1,1,10),person(2,2,10)],[],[], [manages(1,2)]] Snok=[[person(1,1,10)],[],[],[manages(1,1)]]	80%
salary_id	Sok=[[person(1,1,69)],[department(1,1)], [works(1,1)],[]] Snok=[[person(1,1,69)],[department(1,5)], [works(1,1)],[]]	33%

We could view the method for the confidence of the inference mentioned in SubSect. V-B as a game strategy where the goal is to obtain a highest value for the confidence and the gained point. However, two concerns occur with the method: (1) It lacks of a support for the user to specify effectively the domain of snapshots in order to generate examples and counterexamples, and (2) to define the confidence is currently based on the total number of inferred results and the gained point. It needs to complement with a better one, where the input data should be considered, so that the user could better clarify how the inferred result is precise enough.

VI. RELATED WORK

The work in [15] proposes to employ genetic programming in order to automatically discover well-formedness rules in OCL from examples. They encode sets of invariant as populations, where the initial one as instances of a metamodel is produced using invariant patterns. They provide a mechanism with genetic operators in order to evolve the population so that the final generation could accept valid model examples and reject ones. Like our work, that work also employs invariant patterns, but only to produce temporary invariants. Within our approach patterns are used as inference patterns and also as a basis to generate relevant invariants.

There is significant work concentrating on the relationship between OCL constraints and snapshots. In [7] the USE tool allows us to check OCL constraints as restrictions on UML class diagrams. The paper in [22] proposes to validate UML and OCL models by generating snapshots from a declarative description. In [8] the authors propose a method to recheck OCL constraints incrementally as current snapshot changes. Other works often translate the OCL specification into specification environments such as CSP [9], Alloy [10], relational calculus [11], description logic [12] and relational logic [13] in order to effectively validate snapshots or to find a valid (invalid) snapshot or to model-check properties. We

here focus on the other side, how OCL constraints are defined by analyzing snapshots.

Our method to specify OCL invariant pattern is related to several works. The paper in [14] aims to generate OCL constraint template by taking a lexical analysis on input UML models. The paper in [18] proposes a method to represent OCL expressions in a SBVR form in order to generate natural language explanations for business rules. The paper in [23] introduces a method to generate semantically equivalent alternatives for the initially defined OCL constraints. It assists the designer to better define OCL constraints.

The essence of our work is a concept learning, an issue in machine learning. The paper in [24] introduces the L* algorithm in order to generate the best assumption in form of deterministic final-state automata from examples. The paper in [25] proposes a SAT-based method to acquire binary constraint networks. The paper in [19] discusses a method to generate OCL constraints from natural language. Here, we focus on OCL constraints as a target concept to learn.

VII. Conclusion and Future Work

This paper proposes an approach to automating the inference of OCL business rules from User Scenarios. The basic idea of this approach is to employ OCL invariants patterns as inference patterns in order to infer OCL business rules from a set of valid and invalid snapshots. The approach is realized on a pattern-based inference mechanism with the support of the ECL^{i}PS^e solver. In order to strengthen the confidence of the inferred results and to decrease the complexity of the inference process, the paper introduces a method that aims to take advantage of the user's feedback during the inference and to propagate the final inferred result from such a user interaction. The method is realized by a prototype tool and experimented on a running example with the support of a first set of patterns for typical OCL invariants. This indicates the feasibility of applying the proposed approach in practice.

In future, we aim to broaden the case study in order to better evaluate the approach. It could be a task to validate and to infer invariants for the UML metamodel. This requires us to enrich the catalog of patterns as well as to extend the method about eliminating irrelevant invariants and ensuring the confidence of the inference. To enhance the prototype tool with such new features is also on the focus of our future work.

Acknowledgment. This work has been supported by the project ITM-Factory, French FUI 14.

References

[1] A. G. Sutcliffe, N. A. M. Maiden, S. Minocha, and D. Manuel, "Supporting Scenario-Based Requirements Engineering." *IEEE Trans. Software Eng.*, vol. 24, no. 12, pp. 1072–1088, 1998.

[2] D. Hay and K. A. H. A. Healy, *Defining Business Rules ~ What Are They Really?* Business Rules Group, 2013.

[3] OMG, *Semantics of Business Vocabulary and Business Rules v1.1*. OMG, 2013.

[4] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA (2nd Edition)*. Addison-Wesley Professional, 2003.

[5] OMG, *Object Constraint Language OMG Available Specification Version 2.3.1*. OMG, 2012.

[6] R. G. Ross, "RuleSpeak Sentence Forms: Specifying Natural-Language Business Rules in English," *Business Rules Journal*, vol. 10, no. 4, 2009.

[7] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-Based Specification Environment for Validating UML and OCL," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 27–34, 2007.

[8] J. Cabot and E. Teniente, "Incremental Integrity Checking of UML/OCL Conceptual Schemas," *Journal of Systems and Software*, vol. 82, no. 9, p. 1459–1478, 2009.

[9] J. Cabot, R. Clarisó, and D. Riera, "UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models Using Constraint Programming," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE'07, B. F. R. E. Kurt Stirewalt, Alexander Egyed, Ed. New York, NY, USA: ACM, 2007, p. 547–548.

[10] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: A Challenging Model Transformation," in *Model Driven Engineering Languages and Systems*, ser. LNCS, G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, Eds., vol. 4735. Springer Berlin, 2007, pp. 436–450.

[11] L. Mandel and M. V. Cengarle, "On the Expressive Power of the Object Constraint Language OCL," Forschungsinstitut für AngewandteSoftware Technologie, Germany, Technical Report, 1999.

[12] A. Queralt, A. Artale, D. Calvanese, and E. Teniente, "OCL-Lite: Finite Reasoning on UML/OCL Conceptual Schemas," *Data & Knowledge Engineering*, vol. 73, pp. 1–22, 2012.

[13] M. Kuhlmann and M. Gogolla, "From UML and OCL to Relational Logic and Back," in *Model Driven Engineering Languages and Systems*, ser. LNCS, R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds., vol. 7590. Springer, 2012, pp. 415–431.

[14] L. Tan, Z. Yang, and J. Xie, "OCL Constraints Automatic Generation for UML Class Diagram," in *2010 IEEE International Conference on Software Engineering and Service Sciences (ICSESS)*, L. wenzheng, Ed. IEEE Computer Society Press, 2010, pp. 392–395.

[15] M. Faunes, J. Cadavid, B. Baudry, H. Sahraoui, and B. Combemale, "Automatically Searching for Metamodel Well-Formedness Rules in Examples and Counter-Examples," in *Model Driven Engineering Languages and Systems*. Springer, 2013, p. (to appear).

[16] ECLiPSe, *The ECLiPSe Constraint Programming System*. Version 6.1, Jun. 2013. [Online]. Available: http://eclipseclp.org/

[17] B. Demuth, H. Hussmann, and S. Loecher, "OCL as a Specification Language for Business Rules in Database Applications," in *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, M. Gogolla and C. Kobryn, Eds., vol. 2185. London, UK, UK: Springer, 2001, p. 104–117.

[18] R. Pau and J. Cabot, "Paraphrasing OCL Expressions with SBVR," in *Proceedings of the 13th international conference on Natural Language and Information Systems: Applications of Natural Language to Information Systems*, ser. NLDB'08, M. S. E. Kapetanios, V. Sugumaran, Ed., vol. 5039. Berlin, Heidelberg: Springer, 2008, p. 311–316.

[19] I. Bajwa, B. Bordbar, and M. Lee, "OCL Constraints Generation from Natural Language Specification," in *Enterprise Distributed Object Computing Conference (EDOC), 2010 14th IEEE International*. IEEE Computer Society, 2010, pp. 204–213.

[20] M. Richters and M. Gogolla, "On Formalizing the UML Object Constraint Language OCL," in *Conceptual Modeling - ER'98, 17th International Conference on Conceptual Modeling, Singapore, November 16-19, 1998, Proceedings*, ser. LNCS, T. Ling, S. Ram, and M. Lee, Eds., vol. 1507. Springer, 1998, pp. 449–464.

[21] J. Cadavid, B. Combemale, and B. Baudry, "Ten years of Meta-Object Facility: an Analysis of Metamodeling Practices," INRIA, Technical Report RR-7882, 2012.

[22] M. Gogolla, J. Bohling, and M. Richters, "Validating UML and OCL Models in USE by Automatic Snapshot Generation," *Software and System Modeling*, vol. 4, no. 4, pp. 386–398, 2005.

[23] J. Cabot and E. Teniente, "Transformation Techniques for OCL Constraints," *Science of Computer Programming*, vol. 68, no. 3, p. 152–168, 2007.

[24] D. Angluin, "Learning Regular Sets from Queries and Counterexamples," *Information and Computation*, vol. 75, no. 2, p. 87–106, 1987.

[25] C. Bessiere, R. Coletta, F. Koriche, and B. O'Sullivan, "A SAT-Based Version Space Algorithm for Acquiring Constraint Satisfaction Problems," in *Machine Learning: ECML 2005*, ser. LNCS, J. Gama, R. Camacho, P. B. Brazdil, A. M. Jorge, and L. Torgo, Eds., vol. 3720. Springer, 2005, pp. 23–34.